Blockpraktikum: Machine learning physics with language models

Sayan Banerjee and João Sobral

Universität Stuttgart

Institute for Theoretical Physics

v1.1: Last modified on February 10, 2026

Contents

1 Overview

1.1 About this document

This manual contains all the information about the "Blockpraktikum: Machine Learning Physics with Language Models": evaluation criteria, course structure, theory behind the machine learning techniques, and background for the physics problem. Each section is designed to be concise such that it provides only the necessary concepts for each daily problem. Further references are indicated if you want to dive deeper into any of the topics. As for the formatting:

Problem X

This is a general **problem** box.

This is a general **information** box.

As a "lab" course at the University of Stuttgart, this course was designed to involve substantial experimentation. We have intentionally left some open-ended and/or **bonus** questions in this manual, and there are no limits on how much you can explore with the time you have available.

We strongly encourage you (the students) to collaborate as much as possible with each other and the tutors, and to read code documentation and external sources. During the first four days, we will briefly explain the concepts behind the project and the tools you can use—after all, this is a hands-on course. On the last day, you can use the remaining time to work on the project, to address any remaining questions with us (the tutors) or to work on your final lab report.

After the course, you will have to write a lab report with the solutions to each task. If artificial intelligence tools are used to help you during this stage (or prior to it), we expect you to do so responsibly according to the official guidelines from the University of Stuttgart.

1.2 Practical information about the course and evaluation

Location: Seminarraum 5.331, ITP3, Pfaffenwaldring 57

Date: February, 9th–13th, 09:30-12:00.

Final Lab Report Requirements: At the end of the course, each group (composed of 2-3 students) must submit a lab report containing *coherent* solutions to all challenges. While the specific formatting is flexible, we recommend using LaTeX. Please, **concatenate the lab report pdf with all the code you created for the course on a .zip or .rar file**[1].

Report Structure:

- Title "Lab-ML: Group X Final Report" where X stands for your group's number. Include below the names and registration numbers of all group members.
- **Sections 1–4:** One section for each day of the course, containing the corresponding solutions.

[1] Although writing clean and organized code is always a good recommendation, there is no need to refactor your code extensively before submitting the final report.

- (**Bonus section**): Choose one detail that caught your attention during the course (e.g., backpropagation, regularizers, optimizers), do some literature research about it, and explain it in more detail here.
- **Contribution Section:** A final section detailing:
  - How each student contributed to the report and the simulations.
  - If AI tools were used during the project, a description of exactly how they were employed.
- **Conclusion**: Summarize what was done, the key findings, and your interpretation of the results. If applicable, discuss what additional experiments you would conduct to address unexpected results or open questions. You may also suggest potential improvements or extensions to the current approach.

Submission Guidelines:

- **Deadline:** 13/03/2026. This extended deadline is provided to accommodate your other coursework and exam schedules, although we do recommend to write it while everything is fresh in your minds.
- **Format:** .rar or .zip
- **Send to:** joao.sobral@itp3.uni-stuttgart.de
- **Subject line:** "Lab-ML: Group X Final Report", where X stands for your group number.

After submission, we will send you detailed feedback as soon as possible. Please do not hesitate to contact us with any questions before, during, or after the course.

1.3 Prerequisites

We assume that the student has prior *basic* experience with the programming language Python. Furthermore, we also assume some basic knowledge of probability theory, differential calculus, and linear algebra. If necessary, Chapters 3 and 4 from Ref. [1] and Chapter I from Ref. [2] may be useful for reviewing some concepts. In this section, we very briefly review some fundamentals of OOP for Python and provide links for properly setting up Python on your computer (in case you do not have it yet).

Setting up your Python environment First, make sure that Python is installed on your personal computer.[a] An alternative is to use Google Colab. After having your Python setup ready, you can download the course material from the official GitHub repository page. You can then either (i) load it directly into Google Colab or (ii) set up a virtual environment on your local computer. Since (i) is more straightforward, we detail option (ii) as follows. Open your favorite terminal emulator and type:

1. `git clone https://github.com/joaosds/n-lmphysiklab` # download material
2. `cd n-lmphysiklab/` # enter folder
3. `python -m venv lmphysiklab` # example environment name = 'lmphysiklab'

4. `source lmphysiklab/bin/activate` # activate environment
5. `pip install -r requirements.txt` # install necessary Python packages for the course

[a] If not, follow the instructions on the official website to set it up properly depending on your operating system.

Python and OOP Terminology

- **Module:** A python file (e.g., "`test.py`") containing Python code (functions, classes, variables). You can import it in other files to use its contents.

```
# test.py
learning_rate = 0.001          # variable

def square(x):                 # function
    return x ** 2

square_lambda = lambda x: x ** 2  # same function as a lambda
```

- **Class:** A blueprint for creating objects. It defines attributes (data) and methods (functions).

```
class FCNN(nn.Module):
    def __init__(self, input_size):
        super().__init__()         # initialize nn.Module
        self.hidden_dim = 128      # attribute

    def forward(self, x):          # method
        return self.layer(x)
```

- **Object:** Also known as an *instance* of a class. When you create an object, you're making a specific copy from the class blueprint.

```
model1 = FCNN(input_size=784)   # object with 784 inputs
model2 = FCNN(input_size=1024)  # object with 1024 inputs
# model1 and model2 are independent instances with separate attributes

output = model1.forward(x)     # calling the method
```

- **self:** A reference to the current object instance. When you call a method[a], `self` allows the method to access and modify the object's own attributes. For example, `self.hidden_dim` refers to the specific `hidden_dim` of that particular object.

- **__init__:** This *constructor* method runs automatically when you create an object. It initializes the object's attributes and sets up its initial state.

- **super().__init__():** Calls the parent class's constructor. When your class inherits from another class (like `nn.Module`), this ensures the parent class is properly initialized before you add your own customizations. Example of inheritance:

```
class CNN(FCNN):                # inherits from FCNN
    def __init__(self, input_size, num_layers):
        super().__init__(input_size)  # initialize FCNN
        self.num_layers = num_layers  # add new attribute
```

[a] A method is a function defined inside a class, while a function is a standalone block of code.

2 Introduction

> "What I cannot create, I do not understand".
>
> Richard Feynman

From discovering new materials [3] to simulating complex quantum systems [4–11], protein structure prediction [12] and mitigating sources of noise from gravitational wave signals from LIGO [13], artificial intelligence methods have become valuable tools in the natural sciences. Although the general perception of artificial intelligence and its applications has become more widespread only recently[2], some techniques were already being explored in fields like particle physics since the late 1980's [14]. Given this context, it is clear the importance of (i) understanding these techniques and (ii) identifying when they may be relevant for problems we are interested in solving. Understanding comes in different levels, and for us in this short course it will be a heuristic one: we aim at understanding the basic math of neural networks and how to code them (Sec. 1.3), and most importantly how to use them reliably as tools to answer scientific problems (Sec. 4).[3]

The goal of this short course is to teach you how to use some of these techniques for a physical problem. We will create, in a hands-on approach, a simple language model that learns from a non-Markovian generalized Langevin dynamics and generates computationally efficient and quantitatively accurate trajectories. In addition, we will explore quality metrics and interpretability techniques [15] to quantify how well these models "learn" these dynamics. The motivation is then twofold: to demonstrate in practice how n-gram language models can describe non-Markovian dynamics and the role of the parameter n in this description, and to show that by learning how to generate statistically feasible trajectories, one can bypass the explicit solution of a stochastic differential equation with a trained language model, which may be computationally costly in some contexts like coarse-grained molecular dynamics [16, 17].

Instead of starting with large language models using state-of-the-art (SOTA) attention-based archi-

[2] In great deal due to the development and popularization of large language models, and to the Physics and Chemistry Nobel prizes in 2024.

[3] Understanding fundamental principles underlying how large neural networks learn, compute, and scale is an active area of research employing tools from physics, mathematics, computer science, neuroscience, and statistics. For interdisciplinary approaches studying AI as a complex physical system see, for example, the Simons Collaboration on the Physics of Learning and Neural Computation and references in Section 6.

tectures[4] like the Transformer [18], we will use the often neglected, but pedagogically very valuable, n-gram language models with simpler architectures. The complexity of SOTA architectures can obscure the fundamental principles that are important when seeing these topics for the first time, specially in such a short time. As such, the Feynman quote in the beginning of this text serves as a guiding principle, since we seek understanding instead of memorization. By working with simpler models, we aim to build a solid conceptual foundation for you to tackle more advanced methods in the future. Finally, the title of the Blockpraktikum is intentionally general: it emphasizes that the specific (technique) problem could readily be substituted by other (techniques) problems in physics and beyond.

3 A brief introduction to neural networks

We start by giving a high-level meaning to common nomenclature we see everywhere, which at first sight may seem to describe the same thing. **Artificial Intelligence** (AI) can be understood as a broad research and engineering field whose main goal consists of creating intelligent machines which can gather or use knowledge from reality to perform human-like tasks such as understanding speech, images, language, routine labor tasks and even basic scientific research (though how "basic" is still being debated). Importantly, this knowledge may be hard-coded or not. **Machine learning** (ML) is a subset of AI where systems can extract and learn from *raw data*. These methods depend substantially on the current *representation*[5] of the data provided [1]. **Deep learning** (DL) is a subset of ML which uses multi-layered neural networks to learn hierarchical representations of data.

In the next two sections (Sec. 3.1.1 and Sec. 3.1.2) we will explore **supervised learning**, where the model is trained on labeled data to learn underlying patterns and relationships. Once training is complete, the model can generate predictions on new, unlabeled data. In contrast, **unsupervised learning** trains models on unlabeled data, allowing them to discover hidden structures and patterns without explicit guidance. Some examples include clustering and dimensionality reduction techniques, which we will in particular explore in Sec. 3.2. Although we will not cover them in this course, there are also **semi-supervised learning**, which combines small amounts of labeled data with large amounts of unlabeled data (e.g., using a few labeled medical images to improve classification with thousands of unlabeled scans), and a completely different paradigm: **reinforcement learning**, where agents learn optimal behaviors through trial and error by receiving rewards or penalties (e.g., AlphaGo: teaching an AI to play the board game Go [19]).

3.1 Feedforward neural networks

3.1.1 Fully connected neural networks

A first natural step in deep learning is to consider fully connected neural networks (FCNNs), also sometimes called multilayer perceptrons [20, 21]. To a great extent, these networks can be seen as the canonical example of an artificial neural network. Historically, the idea was initiated by the modeling of biological neural networks [22, 23]. Going beyond the analogy with the brain, all "neurons" (nodes) are understood as purely artificial computational units. The general architecture

[4] If you want to learn about these architectures after this course, a great starting point is the deep learning youtube series by 3Blue1Brown.

[5] A representation is a collection of distinct pieces of information or *features* from the data.

of FCNNs is illustrated in Figure 1(a). Information flows in only one direction through the network, with no feedback loops, a defining characteristic of *feedforward* neural networks[6] through a *forward pass.*

Each layer has a certain width, defined by the number of nodes it contains, which are fully connected to the nodes of the adjacent layers through weights, indicated by solid lines. The outer layers consist of an input layer and an output layer. The number of layers determines how deep the neural network is. Between the input and output layers are several hidden layers, whose total number accounts for the depth of the network. Intuitively, the role of the hidden layers is to construct increasingly complex representations of the input in order to solve the given task.

Consider the operation of a single neuron (zoomed-in region of Figure 1(a)). It receives inputs x_i from all nodes in the previous layer and performs two successive transformations. First, it applies an affine linear transformation $y = \sum_{i=1}^{n} w_i x_i + b$. Typically, each node has its own , where bias term b. Second, it introduces nonlinearity through an activation function $\Theta(y)$. This nonlinearity is crucial for enabling the network to build hierarchical feature representations. There are many distinct activation functions, each with unique properties that influence training dynamics (through proper differentiability), computational efficiency (through sparsity [24]), and overall network behavior. Rather than assuming fixed forms, it is worth mentioning that activation functions can also be learned, as it was recently proposed in the context of *Kolmogorov-Arnold neural networks* [25].

To make the last points more concrete, we can consider the simple example of a *perceptron* [20], which uses a Heaviside step activation function

$$\Theta(y) = \begin{cases} 0 & \text{if } y < 0 \\ 1 & \text{if } y \geq 0 \end{cases}.$$

This activation function is well-suited for binary classification, but has limited expressiveness and cannot be used with the backpropagation algorithm due to its non-differentiability[7]. Other typical options include the *rectified linear unit* $\Theta(y) = \max(0, y)$, the *sigmoid function* $\Theta(y) = 1/(1 + e^{-y})$ and the *hyperbolic tangent* $\Theta(y) = \tanh(y)$. For more information, take a look at the PyTorch Reference API, for example.

So far, we have only talked about the building blocks of FCNNs, and what is left is addressing the question: how do we make them *learn*? The answer comes from the pivotal work by Rumelhart, Hinton, and Williams on training neural networks using the backpropagation algorithm [27]. This efficiently allows the computation of gradients of a *loss function* with respect to all network parameters by recursively applying the chain rule from the output layer back through each hidden layer to the input (*backward pass*). The weights w_i and bias vectors b constitute the learnable parameters that are optimized during this training to minimize a loss function $\mathcal{L}(\theta)$ that quantifies the discrepancy between the network's predictions and the true labels, for example, in supervised learning tasks. Each iteration of this optimization routine is called an *epoch*, and its goal is to adjust the parameters closer to values that minimize the loss through the stochastic gradient descent (SGD):

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta \mathcal{L}_B(\theta_t) \tag{1}$$

where θ represents all model parameters (weights and biases), η is the learning rate, and $\nabla_\theta \mathcal{L}_B(\theta_t)$ is the gradient of the loss with respect to the parameters. The subindex B on the loss function

[6] In contrast, neural networks with feedback loops, such as recurrent neural networks, allow information to cycle back through the network, enabling them to handle sequential dependencies more naturally.

[7] The limited expressiveness of perceptrons can also be understood by noting that while they can compute the elementary logical operations $\{\texttt{AND}, \texttt{OR}, \texttt{NAND}\}$ [26], they cannot compute the XOR operation [1].

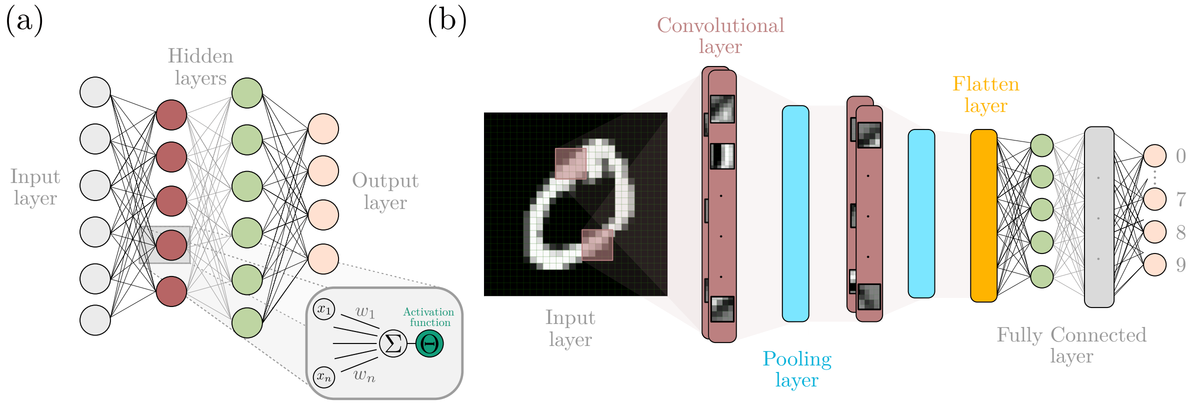

Figure 1: Pictorial representation of the architectures of two feedforward neural networks: the fully connected and the convolutional neural network. (a) For the FCNN, each circle represents an artificial neuron, connected through weights w_i (shown as edges). Each neuron (computational unit) has takes the input $\vec{x}$ also has an associated bias term b_j (not shown) these are indicated separately. Information flows from the input layer through hidden layers to the output layer. (b) For the CNN architecture, the input layer (an MNIST digit image) passes through convolutional layers (pink cylinders) that extract spatial features, a pooling layer (gray) that reduces dimensions, a flatten layer (yellow) that converts features to a vector, and fully-connected layers (green/pink nodes) that perform the final (classification) task.

refers to mini-batches of the training data, which are randomly selected during each iteration. Nowadays, variations of SGD such as Adam [28] are widely used in practice, and developing and analyzing these optimizers is an active research field [29]. Finally, the training process also involves tuning *hyperparameters* including the learning rate η, number of epochs, batch size, and network architecture parameters (e.g., width and depthness for a FCNN).

Vanishing and exploding gradients

A historically significant challenge in training deep networks is the problem of *vanishing* and *exploding gradients* [30, 31]. During backpropagation, gradients are computed by applying the chain rule, which requires multiplying derivatives sequentially across all layers. When these derivatives are consistently smaller than one, gradients shrink exponentially as they propagate backward (vanishing), preventing early layers from learning effectively. Contrastingly, when derivatives are consistently larger than one, gradients grow exponentially (exploding), causing numerical overflow, training instability, and divergent parameter updates. These problems are largely mitigated in modern practice through, for example, careful weight initialization schemes (e.g. Xavier [32]); batch normalization [33], residual connections [34], and gradient clipping techniques [31].

To solidify these concepts through practice, we provide a hands-on Jupyter notebook to be completed with a very classical dataset, the Fashion-MNIST, which contains 70,000 labeled 28 × 28 grayscale images of clothing items across 10 categories. As such, this will be a *classification task*.

Problem 1

1. To begin, import the required PyTorch and torchvision libraries.
2. Get a batch of images from the training loader and visualize them. Are there any classes that look similar?
3. Create a transform that converts images to tensors and split the dataset into:
   - Training set (50,000 samples)
   - Validation set (10,000 samples).
4. Next, we build the full neural network. In PyTorch, the building blocks (linear layers (`nn.Linear`), activation functions (`nn.ReLU`), etc.) are taken care of by the `torch.nn` module. Look at the documentation and fill in the class MLP in the Jupyter notebook.
5. We now define the loss function and optimizer. For the loss function, we use the cross-entropy loss and Adam as an optimizer. Why is cross-entropy loss suitable for this problem?
6. Complete the training loop for a certain number of epochs and run your code. Evaluate the trained model on the test dataset and visualize the predictions.
7. Plot (a) training loss vs epochs and (b) validation loss vs epochs. Do you see underfitting or overfitting? If you do, how would you avoid it?
8. (**Bonus**): Experiment with hyperparameters (learning rate, optimizer etc) and document your observations. For example, try SGD instead of Adam, or add regularizers if you observe overfitting[a].

[a] TensorFlow Playground is an excellent visual aid for building intuition about these parameters before experimenting in your code. Also take a look at Optuna.

Problem 2

In problem 1, you worked with the cross-entropy loss function $H(p,q) = -\sum_x p(x)\log q(x)$ for probability distributions p and q. How does the cross-entropy $H(p,q)$ relate to the KL divergence $D_{KL}(p\|q)$ and the entropy $H(p)$ of the true distribution? Write the relationship and briefly explain its significance for the model training.

3.1.2 Convolutional neural nets

Another important class of feedforward neural networks is the *convolutional neural network* (CNN), which is particularly suited for data with spatial structure, such as images. In 2012, the AlexNet architecture [35] achieved a breakthrough in the ImageNet challenge, significantly outperforming traditional methods in computer vision and demonstrating the power of using graphics processing units (GPUs) for training deep neural networks. Unlike FCNNs where every neuron connects to all inputs, CNNs exploit two key properties of visual data: *locality* (nearby pixels are more relevant than distant ones) and *translation equivariance* (a feature like an edge should be detectable regardless of its position in the image)[8]. The general structure of a CNN is illustrated in Figure 1(b).

The core component of these networks is the *convolutional layer*, which applies learned filters (kernels) to the input layer through convolution operations to produce feature maps. Again, we can take a step back and look at a concrete example to build intuition. The Sobel filter kernels are 3×3 ($N_{\text{kernel}} \times N_{\text{kernel}}$) matrices with fixed entries (weights) that convolve (through the operation $\circledast$) with the source image A_{input} as

$$S_v = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \circledast \mathbf{A}_{\text{input}} \quad \text{and} \quad S_h = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \circledast \mathbf{A}_{\text{input}}.$$

For the example $\mathbf{A}_{\text{input}} = \mathbf{0}$ shown in Figure 1(b), S_v and S_h would produce modified versions of the input with higher intensity for pixels corresponding to vertical and horizontal edges, respectively, from the original image.

The expressivity of CNNs comes from the fact that the kernels are not fixed but *learned* during training, enabling the network to discover task-specific filters that better identify features rather than relying on predetermined patterns. The convolutional operation performs an affine transformation $Y = \mathbf{K} \circledast \mathbf{A}_{\text{input}} + b$, with $\mathbf{K}$ representing the kernel weights (like the matrices shown above) and b the bias term per filter, followed by a nonlinear activation function $\Theta(Y)$ as before.

Each neuron in a convolutional layer is connected only to a small, localized region of the input rather than to all input values simultaneously. The filters then slide across the entire input using the same weights at each position. Through this *weight-sharing* mechanism, translation invariance is achieved: the same filter can detect the same pattern anywhere in the input, whether an edge appears in the top-left or bottom-right corner of an image. This operation enables the network to detect local patterns such as edges and textures in early layers, building up to more complex features in deeper hidden layers.

Since neighboring pixels in images tend to have similar values, the information in convolutional layer outputs may contain some degree of redundancy. Additionally, when performing tasks like classification on image datasets, dimensionality reduction is needed as information is processed through the network for a final output layer with dimension N_{classes}. This is where *pooling layers* come in. A `MaxPooling` layer, for example, retains only the maximum value from each $N_{\text{kernel}} \times N_{\text{kernel}}$ block, while an `AvgPooling` layer computes the average of the values in each block. These pooling operations simultaneously reduce spatial dimensions and introduce additional translation invariance. The extracted features are then typically flattened and passed through fully connected layers that transform them into a format suitable for the final task (e.g. classification for the example shown in Figure 1(b)).

Additionally, although these networks are clearly suited for computer vision tasks, they can also be applied to sequential data such as text and time series [37], as we will see in Sec. 5. But for now, we will get familiar with CNNs by applying them to the previous Fashion-MNIST dataset.

[8] These architectures were also extended to be equivariant with rotation and reflections [36]. For more details, see this nice post by Ed Wagstaff and Fabian Fuchs.

Problem 3

1. To begin, load the data as before. Note that CNN expects images with shape (batchsize, channels, height, width).

2. Complete the CNN model in `class CNN:` with two convolutional layers and a fully connected classifier. Look at the documentation for `nn.Conv2d` and `nn.MaxPool2d` [a] . Note that `MaxPool2d` reduces the spatial size of the image.

3. Create the model, and define the loss function and an appropriate optimizer.

4. Complete the training loop for a certain number of epochs and run your code. Evaluate the trained model on the test dataset and visualize the predictions.

5. Plot (a) training loss vs epochs and (b) validation loss vs epochs. How do these results compare to the ones you obtained with FCNNs?

6. Reduce the size of the training data. Does CNN still perform well? Check the effects of changing the size of the filter (kernel).

7. (**Bonus**): Same as in Problem 4.7. Experiment with hyperparameters (learning rate, optimizer, regularization) and document your observations[b].

8. (**Bonus**): Explain the role of Batch Normalization. When can it prove to be beneficial?

[a] See also [38] for a visualization of how different convolutional layer configurations work in practice.

[b] CNN explainer is another excellent visual aid for building intuition about these parameters before experimenting in your code.

Useful modules from PyTorch for Problem 1

1. `nn.linear(in, width)`: Applies linear transformation $y = Wx + b$, where `in`, `out` refer to the size of input same and output, respectively.

2. `nn.Dropout()`: Dropout randomly sets a fraction of node outputs to zero during each training step. This may help with overfitting.

3. `nn.Sequential()`: Contains all the modules in an ordered fashion, as defined by the user.

4. `nn.CrossEntropyLoss()` : Type of loss function that is useful in training a classification problem.

5. `optim.Adam(model.parameters(), learningrate)`: Optimizes using the Adam algorithm. See also [28] for more details.

Homework: Review SVD and read about PCA for Day 2.

3.2 Principal Component Analysis

One of the simplest, yet powerful examples of unsupervised learning is what is called Principal Component Analysis (PCA). More precisely, it is a linear, unsupervised **dimensionality reduction technique**. In the simplest terms, PCA finds the directions in data that vary the most, rotates the coordinate system to align with them, and throws away the rest. Intuitively, directions with high variance carry the most information about how data points differ from one another: if all samples had nearly the same value along some axis. By retaining only the high-variance directions, PCA discards redundant or noisy dimensions while preserving the *features* that best distinguish samples, allowing to visualize cluster tendencies and/or outliers on datasets and embeddings constructed by a neural network in *latent space*.

In order to explore this concept, let us recap Singular Value Decomposition (SVD) in linear algebra. Simply put, any matrix $M \in \mathbb{R}^{m \times n}$ can be decomposed into three matrices,

$$M = U\Sigma V^T \tag{2}$$

where U, V are orthogonal matrices of sizes $m \times m$ and $n \times n$ respectively and Σ is a diagonal matrix of size $m \times n$. Suppose M represents a centered datasetwhere each of the m rows is a sample and each of the n columns is a feature. The (unnormalized) covariance matrix is then given by

$$M^T M = (U\Sigma V^T)^T (U\Sigma V^T) = V\Sigma^2 V^T. \tag{3}$$

Since PCA finds orthogonal directions in the data that capture maximum variance, and projects the data onto them, the columns of V^T would give the principal directions and Σ^2 would contain the respective variances. Projecting the data onto these principal directions allows us to preserve as much information (variance) as possible.

With this framework in place, let us see how PCA performs in practice with the previously introduced Fashion-MNIST dataset.

Problem 4

1. Load the data as before.
2. Convert images to tensors and flatten them to vectors in 784D space, resulting in an $N \times 784$ matrix where N is the number of images.
3. Center the data by subtracting the mean. Why is this necessary?
4. Calculate the SVD decomposition to extract Σ and V.
5. Project the data onto the first $l = 2$ components of V^T (corresponding to the largest variances) to obtain a low-dimensional representation.
6. Visualize the latent space via a scatter plot in `matplotlib`. Which classes overlap the most and why?
7. Project back to pixel space and compare with the original images. What happens when you increase l?[a]

[a] Check out this visualizer for a nice pictorial representation of PCA.

3.3 (N-gram) language models

The central idea behind **language models** is to assign meaningful probabilities to a certain sequences of tokens $x(t_i) = x_i$ ordered in time $(\cdots < t_{i-1} < t_i < t_{i+1} < \dots)$. More precisely, given a sequence $\vec{x} = (x_1, x_2, \dots, x_k)$, we want to estimate the joint probability $p_\theta(x_1, x_2, \dots, x_k)$, where θ are the model parameters. A good language model should assign high probabilities to sequences that resemble real language and low probabilities to implausible ones. Using the *chain rule* (or general product rule) of probability, we can decompose this joint probability into a product of conditional probabilities:

$$p_\theta\left(x_1, x_2, \dots, x_k\right) = p_\theta\left(x_1\right) p_\theta\left(x_2 \mid x_1\right) \dots p_\theta\left(x_k \mid x_1, x_2, \dots, x_{k-1}\right) = \prod_{i=1}^{k} p_\theta\left(x_i \mid \vec{x}_{<i}\right), \tag{4}$$

where $\vec{x}_{<i} = [x_1, x_2, \dots, x_{i-1}]$. This decomposition is exact, but there is a practical challenge: to predict the i-th token, we would need to condition on the entire preceding context $\vec{x}_{<i}$. For long sequences, estimating these conditional distributions accurably may become intractable[9].

To build any language model, we must take care of three fundamental aspects [37]:

1. (*Modelling*) What functional form does $p_\theta(x_i \mid x_1, \dots, x_{i-1})$ take?
2. (*Learning*) How do we determine the parameters θ from data?
3. (*Inference*) How do we use the model to generate new and meaningful data?

In this section, we will explore ***N*-gram language models**, which arguably give the simplest possible answers to these questions. We will illustrate it in a minimal context using names as a basis for the subsequent exercises on the Langevin equation in the upcoming days.

An **N-gram** is a decomposition of a word or a sentence into smaller fragments of length N. If we take the word "table" as an example, the decomposition into bigrams (2-grams) looks like the set {`ta, ab, bl, le`} and the set of all possible trigrams is given by {`tab,abl,ble`}. Of course, the idea of this decomposition is not restricted to language: any data ordered in time can be modeled similarly. For example, for a particular random walk $X = [x_1, x_2, x_3, x_4]$, where x_i is the position of a particle at a time step i, a bigram decomposition is given by the set $\{[x_1, x_2], [x_2, x_3], [x_3, x_4]\}$ and a trigram separation looks like $\{[x_1, x_2, x_3], [x_2, x_3, x_4]\}$. An N-gram model then assumes an approximation to Equation (4) that the probability for the next step is only dependent on the previous $N - 1$ steps, i.e.,

$$\prod_{i=1}^{k} p_\theta\left(x_i \mid \vec{x}_{<i}\right) \approx \prod_{i=2}^{k} p\left(x_i \mid x_{i-(N-1)}, \dots, x_{i-1}\right). \tag{5}$$

If we take the word "machine" for example, a onegram model would approximate its probability[10] as

$$p(\text{machine.}) = p(\text{m})p(\text{a})p(\text{c})p(\text{h})p(\text{i})p(\text{n})p(\text{e})p(.). \tag{6}$$

[9] Different architectures handle this challenge in different ways, each with inherent advantages and limitations [39]. As we will see, *N-gram models* truncate the context to a fixed window of $N - 1$ tokens. *Recurrent neural networks* compress the entire history into a fixed-size hidden state vector (with fixed dimensionality), updated recursively at each step. Autoregressive *Transformers* [18] build context by attending directly to all previous positions in parallel through the non-local attention mechanism.

[10] We drop the index θ for simplicity from now on.

In contrast, a bigram model would approximate the same word as

$$p(\text{.machine.}) = p(\text{m} \mid \text{.})p(\text{a} \mid \text{m})p(\text{c} \mid \text{a})p(\text{h} \mid \text{c})p(\text{i} \mid \text{h})p(\text{n} \mid \text{i})p(\text{e} \mid \text{n})p(\text{.} \mid \text{e}) \tag{7}$$

Here, we introduce a special character "." to mark the beginning and the end of a word. This will help us to get an estimate of the "size" of names.

To construct these models in practice, we shall use the dataset "Baby Names from Social Security Card Applications – National Data" provided by the SSA. A compiled list of names can also be found at this link.

Problem 5

1. Download the dataset from the link given above. What are some common names?
2. We shall first focus on the 1-gram model.
   (a) Count the number of occurrences of each letter in the dataset. Note that the character "." is also understood as a letter. What is the dimension of your dictionary n_{dictio}?
   (b) Make a histogram to visualize which letters have the most occurrences.
   (c) Calculate the probabilities for each character.
   (d) Generate names by sampling from this probability distribution and stop when you reach the character the dot (".") character.
   (e) Make a list of possible names generated by the 1-gram model. How realistic do they look?
3. Next, we shall do the same but for a 2-gram model.
   (a) First, create a lookup table for each unique character.
   (b) Count the number of occurrences for each letter **pair** in the dataset (i.e., how often each letter follows another letter). Do not forget to take into account the character "." at the start and end of a name.
   (c) Build a $n_{\text{dictio}} \times n_{\text{dictio}}$ correlation matrix and overlay a heatmap to show the probabilities. To calculate the probability distribution, divide each entry of a row by the sum over that row. What are some of the most common letter combinations?
   (d) Generate 100 names by sampling the next letter from the probability distribution based on the current letter.
   (e) Does the 2-gram model perform better than the 1-gram model?
4. (**Bonus**): Build a 3-gram model following the same approach. What are some drawbacks of N-gram models as N is increased?

4 Stochastic Processes

The term *Langevin dynamics* usually refers to the broad modeling of dynamics in molecular systems through stochastic differential equations (SDE), or in simpler words: the modeling of physical objects subject to deterministic and stochastic forces. For example, the *generalized Langevin equation* (GLE), which includes memory effects through time-dependent friction (or memory) kernels, has proven essential for accurately describing non-Markovian protein and barrier crossing dynamics [40, 41].

This kind of dynamics also appears in a pure ML context. Stochastic gradient Langevin dynamics, for example, modifies the standard SGD parameter update (Equation (1)) by adding Gaussian noise with variance proportional to the learning rate, enabling Bayesian posterior sampling for neural networks [42]. Related stochastic processes also play a central role in SOTA **diffusion and score-based generative models**. These models admit complementary descriptions: at the level of individual samples, data evolution can be represented by stochastic or deterministic differential equations, while at the level of distributions, the same dynamics are characterized by partial differential equations (e.g. Fokker–Planck or continuity equations) that govern the time evolution of probability densities [43–45].

Given this context, we will first learn how to create our own simulators for both the standard and generalized Langevin equations. Building on these techniques, our final goal will be to investigate how well simple N-gram language models can capture the physical dynamics governed by the GLE.

4.1 Standard Langevin Dynamics

We start with the SDE

$$m\ddot{x} = -m\gamma\dot{x} + F_{\text{det}} + \eta(t) \tag{8}$$

describing the equations of motion of a particle (in 1D) with mass m in a fluid (or gas) in equilibrium, with friction or damping coefficient $\gamma > 0$ (units of inverse time, $[\gamma] = \text{s}^{-1}$), subject to deterministic ($F_{\text{det}} = \partial U/\partial x$) and random ($\eta$) forces. The friction term $-m\gamma\dot{x}$ describes energy dissipation to the fluid, while the random force $\eta(t)$ takes into account thermal fluctuations arising from collisions of the particle with fluid molecules. Assuming that the fluid is homogeneous and that these collisions have no preferred direction (no systematic bias), the random force can be modeled as Gaussian white noise ("delta-correlated" in time) with zero mean:

$$\langle \eta(t) \rangle = 0 \qquad \text{and} \qquad \langle \eta(t)\eta(t') \rangle = 2\gamma k_B T \delta(t - t'), \tag{9}$$

with k_B representing the Boltzmann constant, T the temperature and the average $\langle \quad \rangle_\eta$ taken with respect to the probability distribution $p_\eta(t)$ of the stochastic variable [46]. The second condition follows from the **fluctuation-dissipation theorem** [47], which ensures that the amplitude of the random force is properly balanced with the dissipation γ, so that the particle reaches thermal equilibrium at temperature T.

Our first task will consist on solving Equation (8) numerically. For this we discretize it in finite time steps Δt with the central difference approximations

$$\dot{x}(t) \approx \frac{x(t+\Delta t) - x(t-\Delta t)}{2\Delta t} \quad \text{and} \quad \ddot{x}(t) \approx \frac{x(t+\Delta t) - 2x(t) + x(t-\Delta t)}{\Delta t^2}. \tag{10}$$

By substituting these relations into the LE (8), one can obtain a recursive relation that gives $x(t + \Delta t)$ in terms of two previous steps $x(t)$ $x(t - \Delta t)$

$$x(t+\Delta t) = \frac{2x(t) - x(t-\Delta t)\,(1 - g(\Delta t)) + \frac{\Delta t}{m}(F + \eta(t))}{1 + g(\Delta t)}. \tag{11}$$

where $g(\Delta t) = \gamma \Delta t/2$. This type of recursion is a form of **Verlet integration**, adapted here to include damping and stochastic forces from the LE (8). The start the recursion, two initial conditions are required, $x(0) = x_0$ and $\dot{x}(t_0) = v_0$. The second initial position can be estimated from these through $x(1) = x(0) + \dot{x}(0)\Delta t$.

Problem 6

Show Equation (11).

Problem 7

1. Complete the code provided to solve Equation (11) for a certain number of position bins N_{bin} in 1D and implement periodic-boundary conditions (PBC) [a]. Plot (i) a few trajectories to check if PBC is implemented correctly and (ii) a histogram of positions at a certain time step t_f after the trajectories have evolved.

2. (**Bonus**) Now consider that the particle is influenced by a double-well potential barrier of the form $U_x = (x^2 - 1)^2$ [41]. Adapt Equation (11) and your code to include this potential and plot again the histogram at a certain time step t_f. How does it differ from your previous results? Compare it with the Boltzmann probability distribution $p(x) = \frac{\exp(-U(x)/k_BT)}{Z}$, where $Z = \int_{-\infty}^{+\infty} \exp(-U(x)/k_BT)$ is the usual partition function.

3. (**Bonus**) Finally, we focus on the deterministic limit ($\eta(t) = 0$) of Equation (8) with a harmonic potential $U = \frac{kx^2}{2}$ to understand the interplay between the friction term $-m\gamma\dot{x}$ and F_{det}.

   (a) Assuming the solution $x(t) = \exp \lambda t$, show that

   $$\lambda = -\frac{\gamma}{2m} \pm \sqrt{\left(\frac{\gamma}{2m}\right)^2 - \omega_0^2} \tag{12}$$

   where $\omega_0 = \sqrt{k/m}$.

   (b) Obtain the analytical solutions $x(t)$ for the underdamped $\gamma < \gamma_c$, critically damped ($\gamma = \gamma_c$) and overdamped ($\gamma > \gamma_c$) cases, where $\gamma_c = 2m\omega_0$.

   (c) Using your code, obtain trajectories in the three regimes for γ with $\eta(t) \neq 0$. Plot the mean value $\langle x(t) \rangle$ as a function of time and compare with the analytical results from item (b). How do they compare?

   (d) A follow-up question to (c): Which variables in your code would be responsible for making these results more accurate?

[a] Introducing these boundary conditions is necessary for keeping a fixed dictionary size during the final project. This step is not necessary for the next two bonus questions, for example

4.2 Generalized Langevin Dynamics

In many complex systems [48], one needs to take memory effects into consideration. In the analogy with language, it is clear from the fact that this sentence can be understood in the context of the previous section that ‘text dynamics’ involves memory: the meaning of words depends on context from earlier sentences retained in our brains. Similarly, the GLE includes *memory kernels* that encode how previous states influence the current state of the particle, unlike the standard LE, where to know a future position one needs to know only the current state. This is why the standard LE is usually called Markovian (or memoryless), whereas the GLE is non-Markovian.

In practice, the GLE is obtained from Equation (8) by introducing a normalized *memory kernel* $M(t)$:

$$m\ddot{x}(t) = -m\gamma \int_0^t M(t-\tau)\dot{x}(\tau)d\tau + F_{\text{det}} + \eta(t). \tag{13}$$

In a real physical system the random force $\eta(t)$ is correlated to the memory kernel via the **second fluctuation-dissipation theorem** $\langle \eta(t)\eta(t')\rangle = 2m\gamma k_B T M(t-t')$ [47]. For simplicity in our simulations, however, we will assume that Equation (9) still holds.

We consider a kernel of the form $M(t) = \omega \exp(-\omega t)$ where large values of $\omega > 0$ suppress long-range memory. More complex kernels may be required to model anomalous diffusion, such as in fractional Langevin dynamics [49], but these cases are beyond the scope of this course.

The time τ integral from $\tau : 0 \to t$ takes into account all (positions) velocities $\dot{x}(\tau)$ up to the final step t. By discretizing the integration of the memory kernel (with the composite trapezoidal rule) using the central difference approximations (10), one can obtain

$$x(t+\Delta t) = \frac{\left[2x(t) - 2\beta(\Delta t)e^{-\omega t}\dot{x}(0) + x(t-\Delta t)\left(\beta(\Delta t)-1\right) - \beta(\Delta t)J(\Delta t) + \frac{\Delta t^2}{m}(F+\eta(t))\right]}{1+\beta(\Delta t)}. \tag{14}$$

where

$$J(\Delta t) = \sum_{k=1}^{n-1} \left(e^{-\omega\Delta t(n-k)}(x((k+1)\Delta t) - x((k-1)\Delta t)\right) \tag{15}$$

where $\beta = \gamma\omega\Delta t^2/2$. Equation (14) should be contrasted with (11). Now, there is a non-Markovian Verlet-type update, in which the next particle position depends on *all* previous steps weighted by the exponential memory kernel.

Problem 8

Show Equation (14).

Composite Trapezoidal Rule [50] Let $[a,b]$ be the interval of integration with $a = x_0 < x_1 < \cdots < x_N = b$ denoting a uniform partition, where $\Delta x = x_j - x_{j-1}$ for all $j = 1, \ldots, N$. The integral of a function $f(x)$ over this interval can be approximated as

$$\int_a^b f(x)dx \approx \Delta x \left(\frac{f(x_0)+f(x_N)}{2} + \sum_{k=1}^{N-1} f(x_k)\right) + \mathcal{O}(\Delta x^2). \tag{16}$$

Problem 9

1. Extend your previous python code to also solve the GLE (13). Plot a few trajectories with distinct values for ω (e.g. $\omega = 0.1, 1, 10$) and compare the histograms of the positions at a certain time step t_f after the trajectories have evolved.
2. For simplicity, set $\omega = 1$ and $\dot{x}(0) = 0$. Now reorganize your code to create a dataset of N_{traj} trajectories for N_{steps} time steps and save it to a *.csv* file. Save your tensor [which should be of shape (N_{traj}, N_{steps})] so that it can be easily loaded later.

5 Final Project

Problem 10

1. (**Bonus**) Construct an n-gram model purely from the dataset of GLE trajectories from Problem 9, i.e., without any neural network implementation. Generate a few trajectories from your n-gram model, plot them as $x(t)$, and compare with the real trajectories. Do they look similar?
2. For the neural network task, start by creating code that loads your dataset from Problem 9 as an input to a neural network architecture. Note that first you will have to pass the data through an `nn.Embedding`(n_{embed}) layer. We recommend starting with a simple MLP as in Problem 1.
3. After training, generate new trajectories and calculate the KL divergence between the true probability distribution $p(x)$ and the learned distribution $p_\theta(x)$ to quantify the similarity:
$$D_{\text{KL}}(p\|p_\theta) = \sum_x p(x) \log \frac{p(x)}{p_\theta(x)}$$
4. Compare your results for a few distinct values of n. Can you see some improvement in the KL divergence as n is increased?
5. Use the PCA technique to understand how your neural network organizes its latent space. Start with $d_{\text{embed}} = 3$. Can anything be said, for example, about the boundary conditions of the trajectories learned by your language model? Next, consider a higher embedding dimension. Can you still infer something from PCA? Whether yes or no, do you have any ideas or intuition about why this happens?
6. (**Bonus**) Take a fixed n and compare the KL divergence results from the MLP to a CNN implementation. Do you see any improvement?

6 What's next?

We have just scratched the surface of what can be done at the intersection of physical sciences and AI. Of course, the challenge (and fun) lies on the fact that one needs to understand concepts coming from a variety of fields [51]. We indicate here a non-exhaustive and biased exemplary list for the interested reader to explore in their own time.

- (2020) Natural Language Processing Course by Lena Voita. *Excellent resource for NLP concepts and historical perspective.*
- (2021) Real-World Natural Language Processing by Masato Hagiwara. *A practical companion to the previous reference.*
- (2023) Neural Networks: Zero to Hero by Andrej Karpathy. *Excellent follow-up series to deepen your understanding of neural networks.*
- (2025) An Introduction to Flow Matching and Diffusion Models by Peter Holderrieth and Ezra Erives. *Excellent introduction to SOTA diffusion-based and score-based generative models. To some extent, our Blockpraktikum can be seen as a preparation for this course.*
- (2019) Machine Learning and the Physical Sciences by Giuseppe Carleo, Ignacio Cirac, Kyle Cranmer, Laurent Daudet, Maria Schuld, Naftali Tishby, Leslie Vogt-Maranto and Lenka Zdeborová. *Overview of applications; a good starting point despite recent advances.*
- (2025) Mechanistic Interpretability for AI Safety by Leonard Bereska and Efstratios Gavve. *A review to mechanistic interpretability methods; see also [52] for current challenges.*
- (2025) Interpretable Machine Learning in Physics by Sebastian Johann Wetzel, Seungwoong Ha, Raban Iten, Miriam Klopotek, Ziming Liu. *A review of interpretability methods applied and developed in physics contexts.*
- (2025) Topics on the mathematics of deep learning by Bruno Loureiro. *Great introduction to a more rigorous mathematical approach to deep learning.*
- (2025) High-Dimensional Learning of Narrow Neural Networks by Hugo Cui. *Statistical physics approach to answer questions about generalization, training dynamics, and sample complexity.*
- (2025) Reinforcement Learning: An Overview by Kevin Murphy. Good companion (alongside [53]) to the classic book [54] on RL.
- (2021) The Principles of Deep Learning Theory by Daniel A. Roberts, Sho Yaida, and Boris Hanin. *Effective theory of deep networks with connections to Bayesian inference and the renormalization group; see [55] for an introductory survey.*

References

[1] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. Adaptive Computation and Machine Learning series. MIT Press, 2016. ISBN: 9780262035613. URL: https://books.google.de/books?id=-s2MEAAAQBAJ.

[2] Kevin P Murphy. *Probabilistic machine learning: an introduction*. MIT press, 2022.

[3] Han Yang et al. "MatterSim: A Deep Learning Atomistic Model Across Elements, Temperatures and Pressures". In: *arXiv* (May 2024). DOI: 10.48550/arXiv.2405.04967. eprint: 2405.04967.

[4] Louis-François Arsenault et al. "Machine learning for many-body physics: The case of the Anderson impurity model". In: *Phys. Rev. B* 90.15 (Oct. 2014), p. 155136. DOI: 10.1103/PhysRevB.90.155136.

[5] Zhenwei Li, James R. Kermode, and Alessandro De Vita. "Molecular Dynamics with On-the-Fly Machine Learning of Quantum-Mechanical Forces". In: *Phys. Rev. Lett.* 114.9 (Mar. 2015), p. 096405. DOI: 10.1103/PhysRevLett.114.096405.

[6] Lei Wang. "Discovering phase transitions with unsupervised learning". In: *Phys. Rev. B* 94.19 (Nov. 2016), p. 195105. DOI: 10.1103/PhysRevB.94.195105.

[7] Juan Carrasquilla and Roger G. Melko. "Machine learning phases of matter". In: *Nat. Phys.* 13 (May 2017), pp. 431–434. ISSN: 1745-2481. DOI: 10.1038/nphys4035.

[8] Yi Zhang, Roger G. Melko, and Eun-Ah Kim. "Machine learning $\mathbb{Z}_2$ quantum spin liquids with quasiparticle statistics". In: *Phys. Rev. B* 96.24 (Dec. 2017), p. 245119. DOI: 10.1103/PhysRevB.96.245119.

[9] Giuseppe Carleo and Matthias Troyer. "Solving the quantum many-body problem with artificial neural networks". In: *Science* 355.6325 (Feb. 2017), pp. 602–606. ISSN: 0036-8075. DOI: 10.1126/science.aag2302.

[10] Joaquin F. Rodriguez-Nieva and Mathias S. Scheurer. "Identifying topological order through unsupervised machine learning". In: *Nat. Phys.* 15 (Aug. 2019), pp. 790–795. ISSN: 1745-2481. DOI: 10.1038/s41567-019-0512-x.

[11] Annabelle Bohrdt et al. "Classifying snapshots of the doped Hubbard model with machine learning". In: *Nat. Phys.* 15 (Sept. 2019), pp. 921–924. ISSN: 1745-2481. DOI: 10.1038/s41567-019-0565-x.

[12] John Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596 (Aug. 2021), pp. 583–589. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03819-2.

[13] Jonas Buchli et al. "Improving cosmological reach of a gravitational wave observatory using Deep Loop Shaping". In: *Science* 389.6764 (Sept. 2025), pp. 1012–1015. ISSN: 0036-8075. DOI: 10.1126/science.adw1291.

[14] Sarah Charley. *Machine learning and experiment*. Symmetry Magazine. Apr. 2024. URL: https://www.symmetrymagazine.org/article/machine-learning-and-experiment?language_content_entity=und.

[15] Sebastian Johann Wetzel et al. "Interpretable Machine Learning in Physics: A Review". In: *arXiv* (Mar. 2025). DOI: 10.48550/arXiv.2503.23616. eprint: 2503.23616.

[16] Maciej Majewski et al. “Machine learning coarse-grained potentials of protein thermodynamics”. In: *Nat. Commun.* 14.5739 (Sept. 2023), p. 5739. ISSN: 2041-1723. DOI: 10.1038/s41467-023-41343-1.

[17] Liyao Lyu and Huan Lei. “Construction of Coarse-Grained Molecular Dynamics with Many-Body Non-Markovian Memory”. In: *Phys. Rev. Lett.* 131.17 (Oct. 2023), p. 177301. DOI: 10.1103/PhysRevLett.131.177301.

[18] Ashish Vaswani et al. “Attention Is All You Need”. In: *arXiv* (June 2017). DOI: 10.48550/arXiv.1706.03762. eprint: 1706.03762.

[19] David Silver et al. “Mastering the game of Go without human knowledge”. In: *Nature* 550 (Oct. 2017), pp. 354–359. ISSN: 1476-4687. DOI: 10.1038/nature24270.

[20] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6 (1958), p. 386.

[21] Marvin Minsky and Seymour A. Papert. *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, 2017. ISBN: 0262534770.

[22] Warren S. McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *Bull. Math. Biophys.* 5.4 (Dec. 1943), pp. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259.

[23] Walter Pitts and Warren S. McCulloch. “How we know universals the perception of auditory and visual forms”. In: *Bull. Math. Biophys.* 9.3 (Sept. 1947), pp. 127–147. ISSN: 1522-9602. DOI: 10.1007/BF02478291.

[24] Ilan Price et al. “Deep neural network initialization with sparsity inducing activations”. In: *arXiv preprint arXiv:2402.16184* (2024).

[25] Ziming Liu et al. “KAN: Kolmogorov-Arnold Networks”. In: *arXiv* (Apr. 2024). DOI: 10.48550/arXiv.2404.19756. eprint: 2404.19756.

[26] Michael A Nielsen. *Neural networks and deep learning*. Vol. 25. Determination press San Francisco, CA, USA, 2015.

[27] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. “Learning representations by back-propagating errors”. In: *Nature* 323 (Oct. 1986), pp. 533–536. ISSN: 1476-4687. DOI: 10.1038/323533a0.

[28] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[29] Dami Choi et al. “On Empirical Comparisons of Optimizers for Deep Learning”. In: *arXiv* (Oct. 2019). DOI: 10.48550/arXiv.1910.05446. eprint: 1910.05446.

[30] Y. Bengio, P. Simard, and P. Frasconi. “Learning long-term dependencies with gradient descent is difficult”. In: *IEEE Trans. Neural Networks* 5.2 (Mar. 1994), pp. 157–166. DOI: 10.1109/72.279181.

[31] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the difficulty of training Recurrent Neural Networks”. In: *arXiv* (Nov. 2012). DOI: 10.48550/arXiv.1211.5063. eprint: 1211.5063.

[32] Xavier Glorot and Yoshua Bengio. “Understanding the difficulty of training deep feedforward neural networks”. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[33] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *arXiv* (Feb. 2015). DOI: 10.48550/arXiv.1502.03167. eprint: 1502.03167.

[34] Kaiming He et al. “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”. In: *arXiv* (Feb. 2015). DOI: 10.48550/arXiv.1502.01852. eprint: 1502.01852.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).

[36] Taco S. Cohen and Max Welling. “Group Equivariant Convolutional Networks”. In: *arXiv* (Feb. 2016). DOI: 10.48550/arXiv.1602.07576. eprint: 1602.07576.

[37] Elena Voita. *NLP Course For You*. Sept. 2020. URL: https://lena-voita.github.io/nlp_course.html.

[38] Vincent Dumoulin and Francesco Visin. “A guide to convolution arithmetic for deep learning”. In: *arXiv* (Mar. 2016). DOI: 10.48550/arXiv.1603.07285. eprint: 1603.07285.

[39] M. Hagiwara. *Real-World Natural Language Processing: Practical Applications with Deep Learning*. Manning, 2021. ISBN: 9781617296420. URL: https://books.google.de/books?id=Ok5NEAAAQBAJ.

[40] Cihan Ayaz et al. “Non-Markovian modeling of protein folding”. In: *Proceedings of the National Academy of Sciences* 118.31 (2021), e2023856118. DOI: 10.1073/pnas.2023856118.

[41] Florian N. Brünig, Roland R. Netz, and Julian Kappler. “Barrier-crossing times for different non-Markovian friction in well and barrier: A numerical study”. In: *Phys. Rev. E* 106.4 (Oct. 2022), p. 044133. DOI: 10.1103/PhysRevE.106.044133.

[42] Max Welling and Yee Whye Teh. “Bayesian Learning via Stochastic Gradient Langevin Dynamics”. In: *Proceedings of the 28th International Conference on Machine Learning (ICML)*. Omnipress, 2011, pp. 681–688. URL: https://www.stats.ox.ac.uk/~teh/research/compstats/WelTeh2011a.pdf.

[43] Yang Song et al. “Score-Based Generative Modeling through Stochastic Differential Equations”. In: *arXiv* (Nov. 2020). DOI: 10.48550/arXiv.2011.13456. eprint: 2011.13456.

[44] Peter Holderrieth and Ezra Erives. *Introduction to Flow Matching and Diffusion Models*. 2025. URL: https://diffusion.csail.mit.edu/.

[45] Michael S. Albergo, Nicholas M. Boffi, and Eric Vanden-Eijnden. “Stochastic Interpolants: A Unifying Framework for Flows and Diffusions”. In: *arXiv* (Mar. 2023). DOI: 10.48550/arXiv.2303.08797. eprint: 2303.08797.

[46] Giuseppe Mussardo. *Statistical Field Theory: An Introduction to Exactly Solved Models in Statistical Physics*. Oxford, England, UK: OUP, Mar. 2020. ISBN: 978-0-19109217-6. URL: https://books.google.de/books/about/Statistical_Field_Theory.html?id=CcjXDwAAQBAJ&source=kp_book_description&redir_esc=y.

[47] R. Kubo. “The fluctuation-dissipation theorem”. In: *Rep. Prog. Phys.* 29.1 (Jan. 1966), p. 255. ISSN: 0034-4885. DOI: 10.1088/0034-4885/29/1/306.

[48] Anatolii V. Mokshin, Renat M. Yulmetyev, and Peter Hänggi. “Simple Measure of Memory for Dynamical Processes Described by a Generalized Langevin Equation”. In: *Phys. Rev. Lett.* 95.20 (Nov. 2005), p. 200601. DOI: 10.1103/PhysRevLett.95.200601.

[49] Eric Lutz. “Fractional Langevin equation”. In: *Phys. Rev. E* 64.5 (Oct. 2001), p. 051106. DOI: 10.1103/PhysRevE.64.051106.

[50] Anosh Joseph. *Markov Chain Monte Carlo Methods in Quantum Field Theories*. Cham, Switzerland: Springer International Publishing, 2020. ISBN: 978-3-030-46044-0. URL: https://link.springer.com/book/10.1007/978-3-030-46044-0.

[51] Andrew Ferguson et al. “The Future of Artificial Intelligence and the Mathematical and Physical Sciences (AI+MPS)”. In: *arXiv* (Sept. 2025). DOI: 10.48550/arXiv.2509.02661. eprint: 2509.02661.

[52] Lee Sharkey et al. “Open Problems in Mechanistic Interpretability”. In: *arXiv* (Jan. 2025). DOI: 10.48550/arXiv.2501.16496. eprint: 2501.16496.

[53] Bruno C da Silva and Philip S Thomas. “COMPSCI 687: Reinforcement Learning Lectures Notes (Fall 2024)”. In: ().

[54] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. Cambridge, MA, USA: The MIT Press, 2018. ISBN: 978-0-262-03924-6.

[55] Daniel A. Roberts. “Why is AI hard and Physics simple?” In: *arXiv* (Mar. 2021). DOI: 10.48550/arXiv.2104.00008. eprint: 2104.00008.